

Client Data Automation Platform

Overview

An intelligent automation system that extracts, validates, and manages client data from multiple sources (HTML forms, email files, and invoice documents) with human-in-the-loop oversight. The platform eliminates manual data entry while maintaining accuracy through supervised approval workflows, centralizing all client and financial data in Google Sheets.

The Business Challenge

TechFlow Solutions processes diverse client data daily from multiple sources:

- Website contact forms (HTML)
- Client inquiry emails (EML files)
- Invoice documents (HTML)

Current State Problems:

- Time-Intensive: 2.5-3.5 hours daily spent on manual data entry
- Slow Response: 24-48 hours to process client inquiries
- Error-Prone: 5-10% error rate in manual transcription
- Not Scalable: Cannot handle volume increases without additional staff
- Fragmented Data: Information scattered across multiple sources

Business Impact:

- Multiple hours per week of staff time consumed by data entry
- Delayed client responses affecting customer satisfaction
- Limited capacity for business growth

The Solution

The TechFlow Client Data Automation Platform is a web-based system that:

- Automatically discovers new documents in configured directories
- Intelligently extracts structured data using AI (OpenAI GPT-4o-mini)
- Validates extracted data for accuracy (email formats, calculations, etc.)
- Presents results in an intuitive dashboard for human review
- Writes approved data directly to Google Sheets for centralized access

Key Features

Intelligent Processing

- AI-powered extraction adapts to document variations
- Rule-based fallback ensures reliability
- Confidence scoring guides review priority
- Comprehensive validation catches errors

Human-in-the-Loop Design

- No data saved without explicit approval
- Easy editing of extracted information
- Side-by-side comparison with source documents
- Reject capability for invalid data

Real-Time Operations

- Instant notifications when new items are ready
- Live dashboard updates via WebSocket
- Batch processing of multiple documents
- Fast response times (<2 seconds per document)

Seamless Integration

- Direct write to Google Sheets
- Organized by data type (Clients/Invoices)
- Complete audit trail with timestamps
- Accessible to entire team

Business Impact

Quantified Benefits

Time Savings

Current: 15-20 minutes/document to check for errors, read everything and upload them manually to Google Sheets. Assuming 10 documents daily for a company of 50-100 employees, that's 150-200 minutes per day, or 2h30m - 3h20m daily. This translates to **52h30m - 70h per month** (assuming 21 working days).

With Automation: 10 documents take around 50 seconds to process. A human would take 2 minutes per document on average to review it and accept it or reject it. In reality, it is less, but document editing is accounted for in the 2 minutes calculation. Thus, for 10 documents

it's 20 minutes and 50sec (=20.83 mins) daily, which translates to 0.347hours/day, totaling **7.29 hours per month** assuming 21 workdays per month.

Savings: Assuming the middle of the value for the current system, 61h15m(=61.25hr), then **53.96 hours are saved per month (88.1% reduction)**

Annual Savings: 735 hours initially to $7.29 \times 12 = 87.48$ hours - **647.52 hours saved**

Cost Savings

Labor savings: With a wage of gross 1,800€/month (aka 1,500€ net in Greece for a period of 12 months), aka 10.71€/hour for 21 workdays/month, **initial costs** are $61.25 \text{hr} \times 10.71\text{€/hr} = 656.25\text{€ every month or } 7,875\text{€ per year}$.

New labor costs are going to be $7.29\text{hr} \times 10.71\text{€/hr} = 78.08\text{€/month or } 936.96\text{€ per year}$, thus **saving 578.17€/month or 6,938.04€/year**.

Operating costs: As of December 2025, gpt-4o-mini costs \$0.15/1M tokens for input and \$0.60/1M tokens for output. Assuming 10 documents per day for 21 working days per month, **210 documents/month** are to be processed. HTML Forms take ~500 tokens input (HTML structure + prompt), ~200 tokens output (JSON with extracted fields), Emails take ~800 tokens input (email headers + body + prompt), ~250 tokens output (JSON with client/invoice data) and Invoices take ~600 tokens input (Invoice HTML + prompt), ~200 tokens output (JSON with financial data). **Document average is ~650 tokens input and ~220 tokens output**, meaning the **monthly number of tokens** is $210 \text{ docs} \times 650 \text{ tokens} = 136,500 \text{ tokens} = 0.1365\text{M tokens (input)}$ and $210 \text{ docs} \times 220 \text{ tokens} = 46,200 \text{ tokens} = 0.0462\text{M tokens output}$. Thus, **monthly cost is** going to be $0.1365\text{M} \times \$0.150 = \0.020 for input and $0.0462\text{M} \times \$0.600 = \0.028 for output, totaling: $\$0.048 \approx \$0.05/\text{month} \approx 0.05\text{€/month}$.

In order to have a safety margin for volume spikes, retries and potential price increases on OpenAI's side, as well as for potential future growth where 10x documents get processed everyday, **2€/month** for OpenAI API at a worst case scenario is gonna be needed, or **24€/year**.

Note: The 2€/month OpenAI API budget is a conservative estimate. Actual usage for 210 documents/month is approximately 0.05€/month. This budget provides a 40-60x safety margin for volume growth, retry logic, and potential price increases, ensuring long-term cost predictability.

Net savings: 576.17€/month (6,914.04€/year)

All the above can be summarized in the following table:

Metric	Manual Process	With Automation	Savings
Monthly Time	61.25 hours	7.29 hours	53.96 hours (88.1%)
Annual Time	735 hours	87.48 hours	647.52 hours

Monthly Labor Cost	656.25€	78.08€	578.17€
Annual Labor Cost	7,875€	936.96€	6,938.04€
Annual Operating Cost	0	24€	-
Net Annual Savings	-	-	6,914.04€

ROI

ROI: ROI = (Net Benefit) / (Cost Of Investment). Therefore, since, as explained in the proposal, development costs are 3,800€, ROI is calculated as follows:

First Year ROI:

Total Investment: €3,800 (development) + €24 (annual operating) = €3,824

Annual Savings: 6,938.04€ (labor saved)

Annual Operating Cost = €24

Net Benefit: 6,938.04€ - 24€ = 6,914.04€

ROI: (6,914.04€ / 3,824€) × 100% = 180.8%

Payback Period = Total Investment / Monthly Savings

Payback Period = 3,824€ / 576.17€ = 6.64 months

Subsequent Years (no development cost):

Investment: 24€/year

Savings: 6,938.04€/year

ROI: (6,914.04€ / 24€) × 100% = 28,808%.

Note: In reality, ROI is gonna be even higher than that, since LLM models are getting cheaper and cheaper.

All the above can be summarized in the following tables:

First Year ROI

Item	Amount
Development Cost	3,800€
Annual Operating Cost	24€
Total Investment	3,824€
Annual Labor Savings	6,938.04€
Less: Operating Cost	-24€

Net Benefit	6,914.04€
ROI	180.8%
Payback Period	6.64 months

Subsequent Years ROI (Years 2-5)

Item	Amount
Development Cost	0€ (already paid)
Annual Operating Cost	24€
Total Investment	24€
Annual Labor Savings	6,938.04€
Less: Operating Cost	-24€
Net Benefit	6,914.04€
ROI	28,808%
Payback Period	6.64 months

5-Year Financial Projection

Year	Investment	Labor Savings	Operating Cost	Net Benefit
Year 1	3,824€	6,938.04€	24€	6,914.04€
Year 2	24€	6,938.04€	24€	6,914.04€
Year 3	24€	6,938.04€	24€	6,914.04€
Year 4	24€	6,938.04€	24€	6,914.04€
Year 5	24€	6,938.04€	24€	6,914.04€
TOTAL	3,920€	34,690.20€	120€	34,570.20€

5-Year Summary

Total investment over 5 years: €3,920

Total savings over 5 years: €34,690.20

Net benefit over 5 years: €34,570.20

Overall 5-year ROI: 882%

Note: In reality, ROI will be even higher as LLM model costs continue to decrease over time.

Quality Improvements:

Error reduction: 90% (from 5-10% to <1%)

Response time: 92% faster (from 24-48 hours to <2 hours)

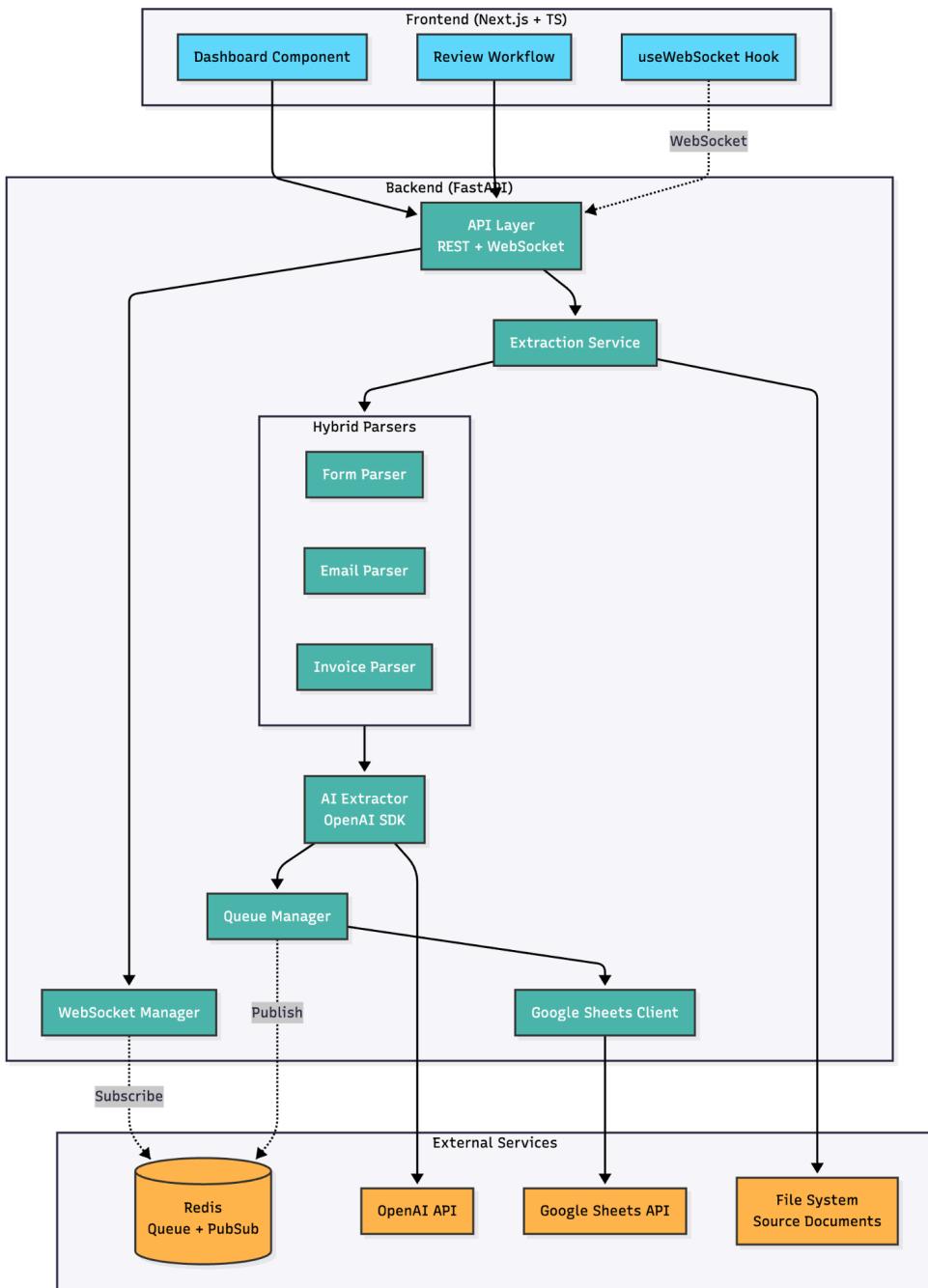
Processing speed: 83.8% faster than manual (from 17.5m on average per document to 2m50s)

Scalability: Can easily handle 10x volume increase without additional staff. Automated processing eliminates bottlenecks.

Technical Aspect

Modern Architecture

Technology Stack



Backend: Python FastAPI (industry-standard for data processing)

Frontend: React (NextJS) + TypeScript (modern, maintainable), Sonner (toast), TanStack Query (initial data fetch with loading/error states, cache invalidation when WS events arrive, automatic retry on failed mutations), shadcn/ui (components) and Tailwind CSS

AI Engine: OpenAI GPT-4o-mini (cost-effective, powerful)

Integration: Google Sheets API (familiar, accessible)

Real-Time: WebSocket (instant notifications and live updates)

Queue & PubSub: Redis (message queue and event broadcasting so the FE can be updated in real-time through the WebSocket)

Infrastructure: Docker (consistent, portable)

Data Flow

Make sure to look into the flow diagram of the project, under the `deliverables/img` folder, for a visual representation of the following data flow.

1. File Discovery & Scanning

- 1.1. User initiates scan via dashboard
- 1.2. POST request to `api/scan` is initiated.
- 1.3. Backend scans configured directories (forms/, emails/, invoices/)
- 1.4. System identifies new, unprocessed files. If none are present, appropriate toast is returned to the user.
- 1.5. Scan operation logged with file count and duration
- 1.6. New files are to have data extracted

2. Intelligent Extraction

- 2.1. Each file is routed to the hybrid parser, which then decides the appropriate parser based on the format of the data (Form/Email/Invoice). The hybrid parser is a **LLM-first parser with rule-based fallback to email and beautifulsoup4 libraries**
- 2.2. AI extraction is attempted first using OpenAI GPT-4o-mini with retries and exponential fallback
- 2.3. Confidence scores are calculated for each extracted field. Confidence score is calculated in 2 stages:
 - 2.3.1. Stage 1 - **AI confidence (70/30 weighting)**. 70% is the average field confidence of the LLM (how sure it is for each extracted field), and 30% comes from the field completeness (how many required fields are present, based on each file's schema). **If the AI confidence here is not high (<70%), the system falls back to the rule-based parser.**
 - 2.3.2. Stage 2 - **Validation Adjustment**. We then decrease this score if there are validation warnings. Errors decrease confidence score by 0.15 each (e.g. invalid email format, incorrect VAT calculation e.g. not being 24% for Greece), while warnings decrease it by 0.05 each (e.g. fields that are missing). I also included different formats of the phone and amount, so it can work seamlessly even if someone enters 1,200.00 or 1200,00, or 69 99999999 or 6999999999.

This 2 stage approach ensures that we account for the inconsistency of the LLM as well as logical errors. High confidence elements can be accepted fast, while low confidence ones need more attention. Confidence is color-coded; green for high-confidence elements, yellow for mid ones, red for low ones. Invoices are very well-defined in terms of schema and formatting, thus are always scored 100%, forms are more or less well-defined too (except for the service type) so they also have very high accuracy, and emails can either contain client data or invoice data, thus field completeness is lower.

- 2.4. Each extraction logged with success/failure status, confidence, and processing time
- 2.5. Errors logged with full context for troubleshooting

3. Queue Management with Redis

- 3.1. Valid extractions are added to a Redis pending queue
- 3.2. Each extraction assigned unique ID and metadata
- 3.3. Redis PubSub publishes "record_added" event and WebSocket broadcasts to all clients (the frontend in our case)
- 3.4. Queue maintains state across multiple users
- 3.5. Queue operations are logged for audit trail

4. Real-Time Notification via WebSocket

- 4.1. WebSocket manager is already subscribed to Redis PubSub events
- 4.2. The frontend shows a toast message that a new file got added and shows the file in real-time.
- 4.3. **Polling could be used** in the stead of WebSockets & PubSub, but a) it wouldn't be a true realtime solution, b) new data wouldn't be added immediately and c) there would be a performance drop since the frontend would be calling the backend every 10 seconds, effectively wasting a lot of API requests, thus increasing latency. **WebSocket connection ensures real-time updates, which means that as soon as new data is extracted and added to the Redis queue, they immediately appear in the dashboard without refreshing the page.**

5. Human Review & Decision

- 5.1. User reviews extraction in dashboard
- 5.2. Confidence score displayed with color coding (green/yellow/red). As mentioned before, high confidence elements can be accepted fast, while low confidence ones need more attention. Therefore, **errors and warnings appear first and the rest of the cards appear in increasing order of confidence score**, so that we immediately look into the ones we're unsure of.
- 5.3. Validation warnings highlighted
- 5.4. User can view the original source document. "View Source Document" gets clicked and calls `api/source/{record_id}` which returns the document and shows it to the frontend with its correct format. text/plain format is the format used for an email, and text/html for forms and invoices.
- 5.5. Filtering based on data type is also possible, so that the user can focus on what is the most important type of document they have to check at the time, e.g. invoices.
- 5.6. Analytics are also present. More specifically:
 - 5.6.1. "Pending Review". How many items are still pending review.
 - 5.6.2. "With Warnings". How many items have warnings.
 - 5.6.3. "Approved". How many items have been approved already in this batch.
 - 5.6.4. "Rejected". How many items have been rejected already in this batch.
 - 5.6.5. "Errors". How many items have errors.
- 5.7. The status of the WebSocket connection appears as a WiFi icon.

- 5.8. Three actions available:
 - 5.8.1. Approve: Proceed to Google Sheets
 - 5.8.2. Edit: Modify fields, then approve
 - 5.8.3. Reject: Remove from queue

6. **Edit Workflow (if needed)**

- 6.1. User clicks Edit button
- 6.2. Modal displays all extracted fields
- 6.3. User modifies incorrect values
- 6.4. PATCH `api/edit/{record_id}` request updates record in Redis queue
- 6.5. Redis PubSub publishes "record_updated" event
- 6.6. WebSocket broadcasts update to all clients
- 6.7. Card refreshes with updated data
- 6.8. Edit action logged with changed fields and user information

7. **Approval & Persistence**

- 7.1. User clicks Approve button
- 7.2. POST `api/approve/{record_id}` retrieves record from Redis queue
- 7.3. Data is written to the appropriate Google Sheets tab:
 - 7.3.1. Client data (forms/emails) → "**Clients**" sheet
 - 7.3.2. Invoice data → "**Invoices**" sheet
 - 7.3.3. An important thing to note here is that **the system routes the data automatically to the correct sheet**, and, furthermore, the **sheets do not even need to be initialized by the user; both the sheets and the columns are handled automatically.**
- 7.4. Retry logic handles API failures (3 attempts with exponential backoff)
- 7.5. On success, record removed from Redis queue
- 7.6. Redis PubSub publishes "record_removed" event
- 7.7. WebSocket broadcasts removal to all clients
- 7.8. Success notification displayed since useWebSocketHook retrieves it, and card is removed from the dashboard
- 7.9. Approval logged with timestamp, user, and Google Sheets row number

8. **Rejection Workflow**

- 8.1. User clicks Reject button
- 8.2. POST `api/reject/{record_id}` is called and the record is immediately removed from Redis queue
- 8.3. Redis PubSub publishes "record_removed" event
- 8.4. WebSocket broadcasts to all clients
- 8.5. Rejection logged with reason and user information for audit purposes
- 8.6. Card removed from dashboard

Key Architectural Benefits

Asynchronous Processing: Redis queue enables non-blocking operations

Multi-User Support: WebSocket broadcasts keep all users synchronized

Fault Tolerance: Retry logic and fallback mechanisms ensure reliability

Scalability: Redis and WebSocket architecture supports horizontal scaling

Comprehensive Logging: Structured JSON logs capture all operations for troubleshooting and compliance

Complete Audit Trail: Every action logged with timestamps, user information, and outcome for full traceability

Design Principles:

- Production-ready from day one
- Scalable architecture with clear growth path
- Comprehensive error handling
- Security best practices
- Extensive documentation, including **Setup Instructions** and a **User Guide**
- Quality Assurance

Automated Testing:

- >80% backend code coverage (see img below), frontend core components thoroughly tested too
- Unit tests for core functionality
- Integration tests for end-to-end workflows
- Pre-commit and .husky hooks prevent broken code (Ruff, pytest, Black for BE, pre-commit ESLint and pre-push Vitest for the FE).
- Continuous testing in development

```

collected 195 items
tests/test_ai_extractor.py .....
tests/test_config.py .....
tests/test_google_sheet.py .....
tests/test_main.py .
tests/test_models.py .
tests/test_queue_manager.py .....
tests/test_rule_based_parsers.py .....
tests/test_utils.py .....
tests/test_websocket_manager.py .....
tests/integration/test_api_endpoints.py .....
tests/integration/test_extraction_service_integration.py .....
tests/integration/test_google_sheets_integration.py .....
tests/integration/test_hybrid_parsers_integration.py .....
tests/integration/test_llm_parser_integration.py .....

----- coverage: platform linux, python 3.11.14-final-0 -----
Name                      Stmts  Miss  Cover
app/__init__.py           1      0   100%
app/api/__init__.py       0      0   100%
app/api/routes.py        149    49   67%
app/config.py             74     2   97%
app/integrations/__init__.py 2      0   100%
app/integrations/sheets.py 127    29   77%
app/logging_config.py    23     3   87%
app/main.py               53     5   91%
app/models/__init__.py    2      0   100%
app/models/extraction.py 49     0   100%
app/parsers/__init__.py   11     0   100%
app/parsers/base.py      11     2   82%
app/parsers/hybrid/__init__.py 4      0   100%
app/parsers/hybrid/email_parser.py 55    11   80%
app/parsers/hybrid/form_parser.py 55    6   89%
app/parsers/hybrid/invoice_parser.py 55    11   80%
app/parsers/llm_based/__init__.py 5      0   100%
app/parsers/llm_based/email_parser.py 34     4   88%
app/parsers/llm_based/extractor.py 114    25   78%
app/parsers/llm_based/form_parser.py 34     4   88%
app/parsers/llm_based/invoice_parser.py 49     7   86%
app/parsers/rule_based/__init__.py 4      0   100%
app/parsers/rule_based/email_parser.py 123    21   83%
app/parsers/rule_based/form_parser.py 47     2   96%
app/parsers/rule_based/invoice_parser.py 124    32   74%
app/parsers/utils.py      31     0   100%
app/pending_queue/__init__.py 4      0   100%
app/pending_queue/manager.py 120    19   84%
app/pending_queue/redis_client.py 15     0   100%
app/pending_queue/websocket_manager.py 54     2   94%
app/services/__init__.py   2      0   100%
app/services/extraction.py 116    2   98%
app/test_parsers.py       46     46  0%

TOTAL                     1593   282  82%
=====
===== 195 passed in 950.69s (0:15:50)

```

Code Quality:

- Automated formatting (Black, Prettier), included in pre-commit hooks and .husky
- Linting (Ruff, ESLint), included in pre-commit hooks and .husky
- Type checking (TypeScript)

Monitoring:

- Structured logging for troubleshooting. Log file with rotation is also created in `backend/logs`
- Health check endpoints
- Error tracking and alerting
- Performance metrics

Remarks

- GitHub Actions could have been a choice for CI/CD. Husky and pre-commit are preferred since it is a single person project, thus 1 development branch is enough too. If more developers are onboarded, CI/CD is strongly recommended for bug prevention and continuous pipeline reliability.
- FE test coverage is something that in a real production use case one doesn't need to take into account thoroughly. Main components being tested is enough, in order not to hinder the development process.

- I added LLM for the extraction so it works even with unexpected formats. However, for a better solution, in a real-world scenario I would be requesting the data in a specific format from my customers, be it through a contact form or through an email. An email template could be enforced on the website side too. This would ensure faster processing without being based on a LLM, that is of course not 100% reliable or consistent. In this context, since rule-based parsing exists as a fallback when confidence is low, parsing is always correct, but to make sure we are future-proofing and optimizing in terms of speed and reliability, strict format guidelines should be enforced. In this case, even if a foreign customer gets onboarded, the models can be updated as expected and the pipeline continues working seamlessly with minimal configuration.
- I didn't grab metrics that were semi-important, such as payment methods. Everything in the template csv is added.
- API calls all have exponential backoff for rate limiting, because the OpenAI API is prone to doing so.
- Make sure to read the USER_GUIDE.md file so you understand all functionalities of the platform.